4Н

УДК 681.3

А.А. Никоненко
Киевский национальный университет имени Т. Шевченко, г. Киев, Украина
andrey.nikonenko@gmail.com

Использование паттернов проектирования в компьютерной лингвистике. Порождающие паттерны. Часть I. Abstract Factory и Builder

Статья посвящена вопросу использования порождающих паттернов в решении задач компьютерной лингвистики. В статье дается определение паттернов, история их создания, рассматривается структура, особенности использования и возможности применения. Подробно разобраны паттерны Abstract Factory и Builder.

Введение

Цель данной статьи – провести исследование, посвященное вопросу применения некоторых приемов построения информационных систем при решении задач компьютерной лингвистики.

С момента своего появления языки программирования постоянно развивались, появлялись новые методы и подходы к созданию программных продуктов. В качестве основных этапов такого развития можно выделить: машинно-ориентированные языки (программирование в машинных кодах, ассемблеры), процедурные языки, объектно-ориентированные языки. Таким образом, можно отметить, что эволюция программных средств протекает в сторону создания более абстрактных инструментов, призванных решать не столько задачи реализации определенного функционала, сколько задачи проектирования системы. С середины 90-х годов на базе объектно-ориентированных языков начинает активно развиваться новый подход к разработке сложных систем – использование паттернов (или шаблонов) проектирования. На сегодняшний день использование паттернов является вершиной эволюции методов разработки программного обеспечения.

Использование паттернов базируется на создании иерархий классов специального вида. Реализация такого подхода не налагает на язык программирования никаких дополнительных требований, а потому поддерживается всеми объектно-ориентированными языками. Широкое распространение подход получил благодаря некоторым своим специфичным свойствам:

1. Позволяет повторно использовать дизайн (а иногда и код) в другом проекте без внесения изменений.

2. Увеличивает гибкость разработанной системы, позволяя вносить или удалять из нее элементы за минимальное время, без переработки системы целиком.

3. Упрощает сопровождение системы, позволяя изменять ее части независимо.

4. Предоставляет легкий способ единообразно конфигурировать и изменять всю систему целиком.

Основная идея паттернов заключается в том, что существуют определенные классы задач, которые программисту приходится решать вновь и вновь при разработке различных приложений. Соответственно, существует некий каталог таких типичных задач и типичных методов их решения. Наиболее известными на сегодняшний день являются несколько таких каталогов или языков (language), как их иногда называют авторы. Это каталог паттернов группы авторов, получившей в интернете название «банда четырех» (gang of four) [1], и каталог «Фаулеровских» паттернов, изложенных в работе [2]. Существуют и другие каталоги паттернов, например, каталог паттернов, рекомендованных для использования в языке Java, составленный специалистами компании Sun [3], и каталог паттернов для .net разработчика [4]. Составлены также каталоги специфичных паттернов и для некоторых других языков программирования. Существуют и каталоги паттернов от крупных ИТ-компаний, например, от Microsoft.

Будучи ограниченными рамками статьи, мы не сможем рассмотреть все популярные каталоги паттернов, поэтому остановимся на классическом каталоге «банды четырех» [1]. Далее мы попытаемся проанализировать паттерны данного каталога и оценить возможности их применения при решении задач компьютерной лингвистики.

Паттерны проектирования

Определение

В разработке программного обеспечения паттерн проектирования (design pattern) – это обобщенное, повторно используемое решение часто встречающейся проблемы проектирования. Паттерн проектирования не является готовой конструкцией, которую можно воплотить прямо в код. Паттерн – это некое описание или шаблон решения проблемы, который может быть применен в различных ситуациях. Объектно-ориентированные паттерны проектирования обычно описывают связи и механизмы взаимодействия классов или объектов, при этом не задавая окончательных классов и объектов приложения, которые будут представлены в коде.

Не все паттерны, применяемые в разработке программного обеспечения, являются паттернами проектирования. Например, алгоритм – это тоже паттерн, но предназначенный для решения вычислительных задач.

История возникновения

Как архитектурный принцип паттерны открыл Кристофер Александер (Christopher Alexander) в 1977 – 1979 годах. Основные идеи подхода он изложил в работе [5]. В 1987 году Кент Бек (Kent Beck) и Вард Каннингам (Ward Cunningham) начали экспериментировать с идеей применения паттернов в программировании. В том же году они презентовали результаты своей работы [6], [7] на конференции OOPSLA, после чего другие исследователи также присоединились к их работе. Результатом совместной работы ученых стала разработка паттернов проектирования для создания графических оболочек на языке Smalltalk.

В 1988 году Эрих Гамма (Erich Gamma) в цюрихском университете начинает работу над докторской диссертацией на тему применения паттернов в проектировании программного обеспечения.

В 1989 – 1991 годах Джеймс Коплин (James Coplien) работает над сходной идеей – разработкой идиом для программирования на C++ и публикует в 1991 году книгу «Advanced C++ Idioms» [8].

В 1991 году Эрих Гамма заканчивает работу над докторской диссертацией и переезжает в США, где в сотрудничестве с Ричардом Хелмом (Richard Helm), Ральфом Джонсоном (Ralph Johnson) и Джоном Влиссидсом (John Vlissides) начинает работу над книгой, посвященной паттернам проектирования. Именно после выхода книги «Design Patterns: Elements of Reusable Object-Oriented Software» [1] в 1994 году паттерны проектирования получают широкую популярность в компьютерных науках. Также в 1994 году проходит первая конференция «Pattern Languages of Programming Conference» и создается Портлендовский Репозиторий Паттернов (Portland Pattern Repository) для документирования паттернов проектирования.

Особенности использования

Использование паттернов проектирования позволяет ускорить процесс разработки ПО, благодаря использованию испытанной, доказанной парадигмы разработки. Построение правильного дизайна информационной системы требует рассмотрения большого количества вопросов, некоторые из которых могут возникнуть только на этапе внедрения. Повторное использование паттернов проектирования при разработке дизайна помогает избежать неверных действий, которые ведут к возникновению проблем на поздних стадиях реализации, а также повышает прозрачность и читаемость кода для других разработчиков и архитекторов, знакомых с паттернами.

Отрицательные моменты применения подхода следующие:

1. Для достижения гибкости паттерны проектирования часто вносят в дизайн дополнительные косвенные уровни, которые в некоторых случаях могут усложнить дизайн и ухудшить производительность приложения.

2. По определению, паттерн должен программироваться заново для каждого приложения. Поэтому некоторые авторы усматривают, что это шаг назад по отношению к повторному использованию кода, предоставляемому компонентами. Поэтому исследователи работают над преобразованием паттернов в компоненты. Мейер (Meyer) и Арноут (Arnout) заявляют о достижении двух третей успеха в компонентизации наиболее известных паттернов [9].

Часто люди понимают, как использовать определенную методологию проектирования для решения определенного класса проблем. Методологии такого вида очень сложно применять для решения более широкого, чем они изначально предназначены, класса задач. В отличие от таких методологий, паттерны проектирования сразу предоставляют общее решение, описанное в формате, не требующем специфических привязок к конкретной проблеме.

Структура

Паттерны проектирования скомпонованы в несколько секций. Всего рассматриваемая нами книга содержит 23 паттерна: Порождающие паттерны (5), Структурные паттерны (7) и Паттерны поведения (11). Каждая из секций предназначена для решения задач определенного вида и содержит прототип микроархитектуры, который разработчик может скопировать и адаптировать к своему собственному дизайну для решения повторяющихся проблем, описанных паттерном проектирования. Микроархитектура – это набор программных компонент (классы, методы и т.д.) и их взаимосвязей. Разработчики используют паттерны проектирования, привнося в свой дизайн эту микроархитектуру, в результате их дизайн будет иметь архитектуру, схожую с описанной в выбранном паттерне.

Дополнительным преимуществом паттернов является упрощение коммуникации между разработчиками, т.к. позволяет им использовать одинаковый словарь терминов при обсуждении архитектуры приложения.

Порождающие паттерны

В связи с ограниченным объемом статьи в первой части мы рассмотрим только возможные применения в лингвистике только нескольких порождающих паттернов. Остальные паттерны будут рассмотрены в следующих статьях. Далее в тексте главы будут использованы некоторые схемы и цитаты из работы [1].

Основная задача, решаемая порождающими паттернами – абстрагирование процесса порождения объектов. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов.

Для порождающих паттернов актуальны две темы. Во-первых, эти паттерны инкапсулируют знания о конкретных классах, применяемых в системе. Во-вторых, скрывают механизмы порождения и взаимодействия экземпляров этих классов. Все, что известно системе об объектах, – это их интерфейсы, объявленные в абстрактных классах. Следовательно, порождающие паттерны предоставляют значительную гибкость в вопросах, что создавать, как создавать и когда создавать.

Существует два стандартных способа параметризации системы классами создаваемых ею объектов. Первый способ – порождение подклассов от класса, создающего объекты; что соответствует паттерну Factory Method. Основной недостаток метода: может понадобиться создавать новый подкласс лишь для того, чтобы изменить класс продукта. Такие изменения могут быть каскадными. Например, если создатель «продукта» сам создается фабричным методом, то придется замещать и его создателя тоже.

Другой способ параметризации системы больше зависит от композиции объектов: определите объект, ответственный за знание классов объектов-продуктов, и сделайте его параметром системы. Это ключевой аспект паттернов Abstract Factory, Builder, Prototype. Все три содержат создание нового «фабричного объекта», ответственного за создание объекта-продукта. В Abstract Factory фабричный объект производит объекты разных классов. Фабричный объект паттерна Builder пошагово создает сложный продукт, следуя специальному протоколу. Фабричный объект паттерна Prototype изготавливает продукт путем копирования объекта-прототипа. В последнем случае фабричный объект и прототип – это одно и то же, поскольку именно прототип отвечает за возврат продукта.

В некоторых случаях сразу несколько порождающих паттернов могут предлагать решение задачи проектирования. В каждом конкретном случае решение о выборе оптимального паттерна должно приниматься отдельно, в зависимости от задач, стоящих перед разрабатываемой системой. Также бывают ситуации, в которых оправдано использование композиции порождающих паттернов.

Паттерн Abstract Factory (Абстрактная фабрика)

Назначение

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Суть

Все обращения клиента идут к одному абстрактному классу (Abstract Factory), который на самом деле реализован одним из конкретных классов-фабрик (Concrete Factory), который умеет производить специфичные для него виды продуктов (т.е. классов). Сам же

клиент обращается к продукту, используя интерфейс, заданный в абстрактном классе данного вида продукта. Схематически паттерн изображен на рис. 1.

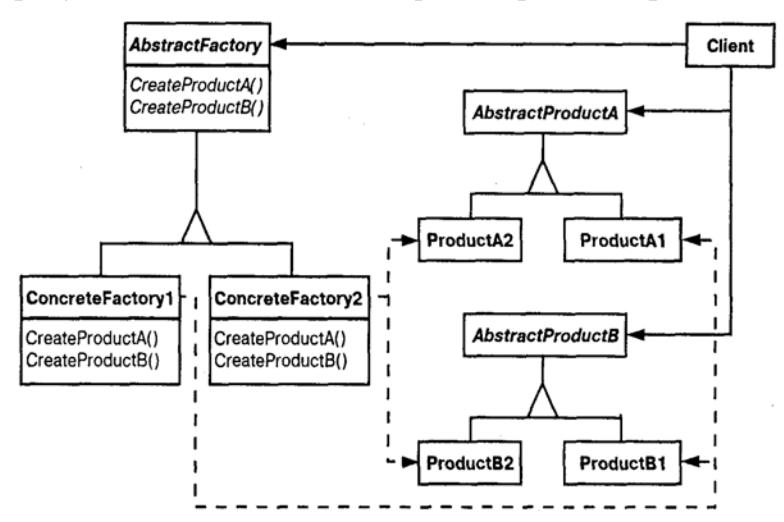


Рисунок 1 – Структура паттерна Abstract Factory

Описание функциональности

Когда приложению требуется породить экземпляр определенного класса, оно не порождает его напрямую, вызывая конструктор нужного класса. Вместо этого приложение обращается к специальному объекту-фабрике с просьбой породить объект нужного класса. Объект-фабрика имеет интерфейс, описанный в абстрактном классе Abstract Factory. В его интерфейс включены методы порождения всех необходимых нам объектов (CreateProductA, CreateProductB…). Таким образом, мы уменьшаем количество зависимостей, заменяя зависимость приложения от всех необходимых ему классов на зависимость от объекта-фабрики. Вызов метода фабрики возвращает нам объект запрошенного класса.

Сами объекты-продукты содержат интерфейсы, описанные в абстрактных классах продукта данного вида и набор подклассов для порождения объектом-фабрикой. Обычно каждый конкретный экземпляр абстрактной фабрики порождает экземпляры своего подкласса продукта. То есть если у нас в системе содержится две реализации абстрактной фабрики (ConcreteFactory1, ConcreteFactory2) и существует два продукта (AbstractProductA, AbstractProductB), то для каждого подкласса фабрики у нас должен существовать свой подкласс продукта. То есть ProductA1 и ProductB1 для ConcreteFactory1 и ProductA2 и ProductB2 для ConcreteFactory2.

В этом заключается неудобство паттерна Abstract Factory – рост иерархии классов. Обычно каждый класс продукта должен иметь столько подклассов, сколько подклассов фабрики существует в системе. То есть при добавлении нового подкласса Abstract Factory нам нужно будет добавить по одному подклассу к каждому классу продукта.

Класс Abstract Factory содержит набор абстрактных методов, которые должны быть реализованы в его подклассах (ConcreteFactory1, ConcreteFactory2, ...). В принципе в методы класса Abstract Factory можно поместить поведение по умолчанию. Экземпляр одного из подклассов абстрактной фабрики порождается в начале работы приложения и, соответственно, конфигурирует приложение подклассами продуктов своего вида. В то же время существует возможность заменить его в любой момент работы приложения на экземпляр другого подкласса фабрики и, таким образом, переконфигурировать все приложение.

Именно легкость конфигурирования системы целиком является основным преимуществом паттерна Abstract Factory.

Использование в компьютерной лингвистике

Наибольшую пользу Abstract Factory может принести в решении вопроса мультиязычности. Самое очевидное его применение в этом контексте – это реализация многоязыковых пользовательских интерфейсов (GUI). В этом случае нам понадобится создать по подклассу Abstract Factory для каждого поддерживаемого языка интерфейса. Сам Abstract Factory должен содержать операции создания всех графических элементов приложения, они и будут выступать продуктами фабрики. Также понадобится создать по подклассу каждого продукта для каждого поддерживаемого языка.

Теперь при выборе пользователем другого языка интерфейса мы будем создавать соответствующий этому языку подкласс фабрики и вызывать процедуру перерисовки окна приложения, все остальное фабрика сделает автоматически.

Описанное выше применение паттерна, безусловно, полезно, однако нас в первую очередь будут интересовать функциональные применения шаблонов проектирования. Предположим, что мы разрабатываем приложение для анализа текстов. Приложение должно обладать следующей функциональностью: определять тематику текста; выбирать из текста информацию по семантическому шаблону; реферировать текст; осуществлять перевод текста на другой язык.

Каждую операцию мы вынесем в отдельный класс. Так, операции по определению тематики будет выполнять класс ThematicAnalyzer, выбирать информацию будет InfoExtractor, строить рефераты – AbstractMaker, а переводить текст – Translator. Если нам понадобится добавить новую функциональность к системе, то мы просто добавим еще один класс, например, SpellingChecker, для поиска ошибок в тексте.

Возникает вопрос, какое архитектурное решение нам нужно применить, если мы хотим, чтоб программа была способна работать с текстами на нескольких языках, например, на русском и английском? При этом подразумевается, что поведение описанных выше классов должно отличаться при обработке текстов на разных языках. Решение – применить паттерн Abstract Factory. В результате дизайн системы примет вид, показанный на рис. 2.

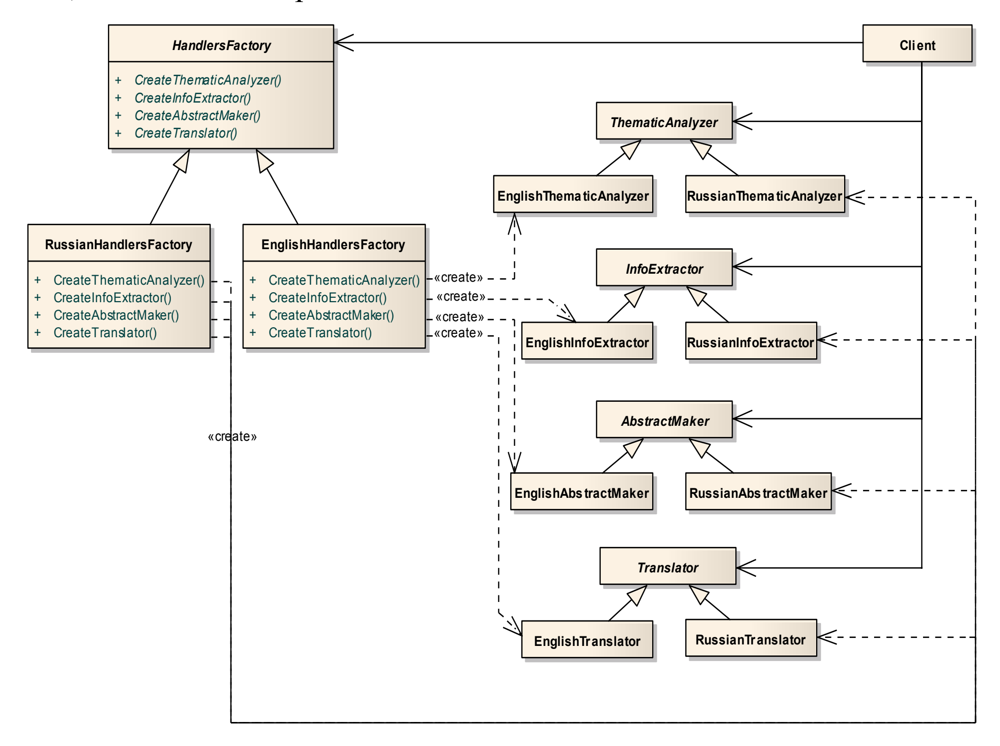


Рисунок 2 – Применение паттерна Abstract Factory в приложении

Все обращения Клиента (в данном случае клиентом выступает наше лингвистическое приложение) идут к фабрике обработчиков (HandlersFactory). То есть если пользователь запросит построение реферата, то система обратится к HandlersFactory и вызовет операцию порождения объекта AbstractMaker (HandlersFactory →CreateAbstractMaker()).

В начале работы с системой пользователь указывает, тексты на каком языке он будет обрабатывать, например, на русском. В результате такого конфигурирования системой будет порожден объект RussianHandlersFactory. И все запросы клиента будут поступать к нему, а не к абстрактной HandlersFactory. А значит, результатом операции CreateAbstractMaker(), описанной в предыдущем абзаце, в данном случае будет объект RussianAbstractMaker. Если же пользователь сменит язык обрабатываемых текстов на английский, то результатом операции будет объект EnglishAbstractMaker.

И последний вопрос, насколько сложно нам будет добавить к уже готовой системе новый язык, например, украинский?

Для добавления нового языка нам не нужно будет переписывать существующий код. Мы добавим еще один подкласс HandlersFactory – UkrainianHandlersFactory и добавим по подклассу для обработки украиноязычных текстов в каждый из обработчиков. То есть UkrainianThematicAnalyzer, UkrainianInfoExtractor, UkrainianAbstractMaker, UkrainianTranslator. После этих действий наша система сможет обрабатывать также и тексты на украинском языке.

Применимость

Используйте паттерн Абстрактная фабрика, когда:

– система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;

– входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения;

– система должна конфигурироваться одним из семейств составляющих ее объектов;

– вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Паттерн Builder (Строитель)

Назначение

Отделяет конструирование составного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

Суть

Абстрагирует процесс конструирования составного (сложного) объекта. Все методы построения частей (BuildPart1, BuildPart2 …) сложного объекта принадлежат одному абстрактному классу (Builder). Абстрактный строитель имеет некоторое количество конкретных реализаций (ConcreteBuilder1, ConcreteBuilder2 …).

Распорядитель (Director) управляет процессом построения сложного объекта, вызывая методы Builder->BuildPart1, Builder->BuildPart2…

Клиент вначале создает объект Builder одного из конкретных классов (ConcreteBuilder), потом создает объект Director, сконфигурированный этим Builder'ом. Builder выполняет все действия автоматически и клиент забирает у него результат.

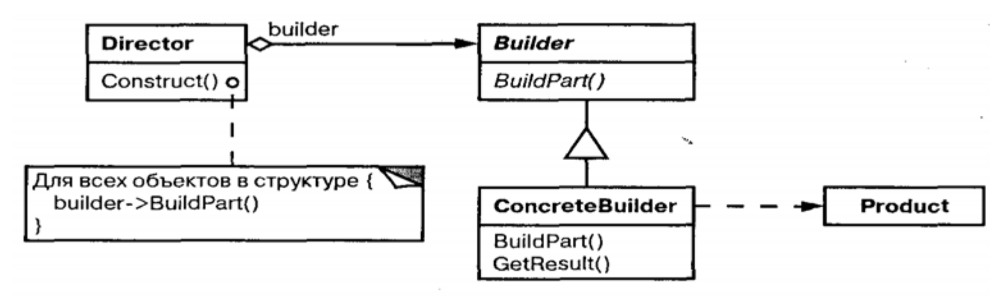


Рисунок 3 – Структура паттерна Builder

Описание функциональности

В целом паттерн Builder похож на Abstract Factory, только теперь производство продукта состоит из частей. Существуют специальные строители (ConcreteBuilder), каждый из которых умеет производить продукт своего вида и содержит все операции по созданию частей своего продукта. Также существует специальный объект-управитель (Director), который руководит производством продукта, задавая последовательность его частей. Общий интерфейс строителей, доступный Director'у, задается абстрактным классом Builder.

Когда клиенту требуется продукт, он создает объект Director, конфигурируя его нужным ConcreteBuilder и передавая данные для построения продукта. Director, в соответствии со своим алгоритмом, обрабатывает исходные данные, передавая запросы строителю. Такой подход позволяет использовать один и тот же алгоритм построения для создания продуктов различного вида. По окончанию работы управителя клиент забирает результат у строителя.

Основные плюсы паттерна: дает более тонкий контроль над процессом создания продукта, т.к. продукт создается по частям, а не весь сразу, как это было в Abstract Factory. Разделяет код, производящий построение продукта, и код, управляющий процессом строительства.

Использование в компьютерной лингвистике

В компьютерной лингвистике паттерн разумно использовать при решении любой задачи, в которой текст рассматривается как составной объект. В качестве примера рассмотрим построение автоматического переводчика.

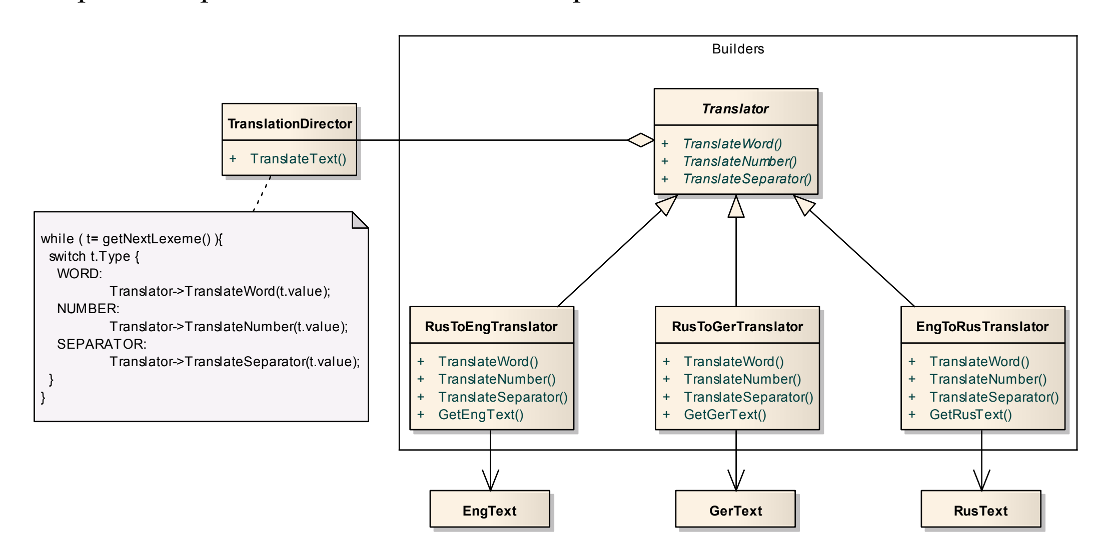


Рисунок 4 – Применение паттерна Builder в приложении

Поскольку с точки зрения перевода текст является составным объектом (содержит слова, числа, разделители), то разумно для его обработки использовать паттерн Builder. В данном случае управитель (TranslateDirector) будет осуществлять разбор текста по лексемам и для каждого типа лексемы вызывать соответствующую ей функцию строителя (Translator). Подклассы строителя будут отвечать за преобразование лексемы в нужный язык.

Предположим, мы строим систему автоматического перевода с русского на английский и немецкий языки. Тогда в качестве одного из подклассов строителя будет использоваться строитель англоязычных текстов (RusToEngTranslator), а в качестве другого – строитель текстов на немецком языке (RusToGerTranslator). Продуктом этих строителей будет выступать текст на соответствующем языке (EngText или GerText).

Такой дизайн позволяет легко добавлять новые подклассы строителя для перевода текстов на другие языки. При этом алгоритм разбора текста, заданный в TranslateDirector, остается неизменным. Если мы хотим добавить в систему возможность перевода текстов с английского на русский, то нам будет достаточно добавить еще один подкласс строителя – EngToRusTranslator. В качестве продукта этот строитель будет возвращать тексты на русском языке.

Применимость

Используйте паттерн Строитель, когда:

– алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;

– процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Литература

1. Design Patterns: Elements of Reusable Object-Oriented Software / [Gamma Erich, Richard Helm, Ralph Johnson and John Vlissides]. – Addison-Wesley, 1995.
2. Fowler Martin. Patterns of Enterprise Application Architecture / Fowler Martin. – Addison-Wesley, 2002.
3. Deepak Alur. Core J2EE Patterns: Best Practices and Design Strategies / Deepak Alur, John Crupi and Dan Malks. – [2nd Edition]. – Prentice Hall / Sun Microsystems Press, 2003, June.
4. Christian Thilmany. .NET Patterns: Architecture, Design, and Process / Christian Thilmany. – Addison-Wesley Professional. – 2003. – August 18. – 448 p.
5. Christopher Alexander. A Pattern Language / Christopher Alexander, Sara Ishikawa, Murray Silverstein [и др.]. – New York : Oxford University Press, 1977.
6. Smith Reid. Panel on design methodology / Smith Reid // OOPSLA '87 Addendum to the Proceedings. OOPSLA '87 (October 1987). – Режим доступа : http://dx.doi.org/10.1145/62138.62151
7. Beck Kent. Using Pattern Languages for Object-Oriented Program / Beck Kent, Ward Cunningham // OOPSLA '87 workshop on Specification and Design for Object-Oriented Programming. OOPSLA '87 (September 1987). – Режим доступа : http://c2.com/doc/oopsla87.html
8. James O. Coplien. Advanced C++ programming styles and idioms / James O. Coplien. – Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1991.
9. Meyer Bertrand. Componentization: The Visitor Example / Bertrand Meyer, Karine Arnout // IEEE Computer (IEEE) 39 (7): 23 – 30. (July 2006). – Режим доступа : http://se.ethz.ch/~meyer/publications/computer/visitor.pdf

A.A. Nykonenko

The Use of Design Patterns in Computer Linguistics. Creational Patterns.
Part I. Abstract Factory and Builder

The article is dedicated to the issue of the use of the creational patterns in computer linguistics problem solving. There are definitions of patterns, history of their creation, structure, features of usage and application abilities in the article. Abstract Factory and Builder patterns are discussed in detail.

Статья поступила в редакцию 13.07.2010.